UFR des Sciences
CAMPUS DE VERSAILLES

Rapport PPN

Rapport de Projet

Réalisé par :
Julien RIGAL
Penda THIAO
Estelle OLIVEIRA Patrick Nsundi MBOLI

Calcul Haute Performance et Simulation
Université de Versailles Saint-Quentin-en-Yvelines

Encadrants:
Hugo Bollore: hugo.bollore@uvsq.fr
Jäsper Salah IBNAMAR : mohammed-salah.ibnamar@uvsq.fr

Date de soumission :
02 Mai 2025

Contents

Abstract

Ce rapport détaille l'architecture et les fonctionnalités principales du projet C++ PPN_CTS, un système de simulation de trading décomposé en plusieurs modules. Il explore les rôles et responsabilités du serveur central, l'interaction des clients, l'automatisation du trading via des bots, ainsi que la gestion des données transactionnelles et financières au sein des portefeuilles.

1 Introduction

Ce second semestre a été pour nous l'occasion de consolider ce que nous avions fait lors du premier, de remanier certaines parties, certaines implémentations, avant d'enfin passer à la partie la plus exigeante : passer d'un modèle séquentiel à un modèle parallèle, et en étudier la scalabilité. Les travaux se sont donc tous axés autour de deux axes principaux : la transition d'une architecture séquentielle à une architecture parallèle, ainsi que la recherche d'efficacité lors de l'implémentation. Chaque partie de ce rapport abordera ces deux thèmes, et sera conclue par une ou plusieurs mesures concrètes de cette efficacité.

Nous commencerons par l'implémentation du serveur, puis parlerons en détail des interactions, parfois nouvelles, entre le client et ce dernier. Enfin, nous conclurons par une partie sur les bots de trading, et une possible ouverture.

2 Le Serveur

Le **Serveur** est par essence à la base de l'architecture. Il met en place l'environnement, est derrière chaque interaction, doit assurer le bon déroulement ainsi que l'orchestration des services internes. Le contexte de multi-threading s'est donc répercuté sur sa conception.

2.1 Responsabilités Clés du Serveur

Tout d'abord, il y a bien sûr l'initialisation du réseau (sockets), configuration et gestion du contexte SSL. Pour cela, les changements ne sont pas majeurs, le principe restant le même que pour une version séquentielle. Nous avons tout de même ajouté une Autorité de Certification (CA), qui est comme une source de confiance tierce.

Ensuite, il y a l'activité *continue* du serveur. Elle pourrait être résumée en quelques points :

- **Acceptation des Connexions** : Boucle principale d'écoute et d'acceptation des nouvelles connexions TCP/SSL entrantes. Cette tâche est déléguée à un thread dédié (`acceptThread`) exécutant la méthode `AcceptLoop` pour ne pas bloquer le fil d'exécution principal.
- **Orchestration des Sessions Clientes** : Pour chaque nouvelle connexion acceptée, le serveur lance un nouveau thread qui exécute la méthode `HandleClient`. Ce thread est responsable de la gestion complète d'une session cliente unique, depuis l'authentification jusqu'à la déconnexion.
- **Gestion Centralisée des Clients Actifs** : Le serveur maintient une liste (`activeSessions`) de toutes les sessions clientes actuellement connectées et actives. Il doit pouvoir ajouter (`HandleClient`) et retirer (`unregisterSession`, appelé par la session elle-même lors de l'arrêt) des sessions de cette liste de manière thread-safe.
- **Gestion des Utilisateurs et Authentification** : Le serveur stocke les informations d'identification des utilisateurs (`users` map, avec mots de passe hachés et salés). Il gère le chargement et la sauvegarde de ces données (`LoadUsers`, `SaveUsers`) et implémente la logique de vérification ou d'enregistrement lors d'une demande d'authentification (`processAuthRequest`), généralement appelée par la `ClientAuthenticator` dans le thread `HandleClient`.

Pour finir, nous pouvons mentionner son rôle clef concernant la gestion du Cycle de Vie : Fournir des mécanismes pour démarrer (`StartServer`) et arrêter (`StopServer`) l'ensemble du système serveur de manière ordonnée, incluant l'arrêt des threads. Et tout cela en garantissant une non-corruption des données.

2.2 Défis d'Implémentation en Multi-Threading

Les défis apportés par une telle implémentation sont principalement liés aux enjeux de concurrence, de ressources partagées, mais il ne faudrait pas omettre la modularité que nous avons dû mettre en place.

En effet, avant même d'arriver à cette collision des threads pour un accès à ces sections critiques, il faut encore faire en sorte que chacun puisse exister par lui-même, et ait des moyens de s'identifier. C'était là l'objet de notre toute première implémentation : **Le stockage, et la distribution des données.**

2.3 Principe du Stockage de Données

Nous avions auparavant une façon naïve de procéder. En effet, lorsqu'un client se connectait, le serveur sauvegardait tout un nombre d'informations dans divers fichiers `csv` le concernant. Et quand les bots du client souhaitaient accéder à l'état du marché, il fallait également aller chercher des valeurs contenues dans d'autres fichiers stockés *dans le disque*.

Étant bien au fait de l'écart de performance entre le fait d'aller chercher des données dans ces fichiers, et aller les chercher dans la mémoire vive, et conscients que notre serveur était voué à gérer plusieurs centaines voire milliers de clients, nous avons opté pour une gestion des données plus efficace.

Les données en flux continu, comme le prix du SRD-BTC, sont maintenant stockées dans un tableau *glissant* qui ne peut contenir qu'un nombre fini de valeurs. L'accès à ces données est donc bien plus rapide. Ces dernières sont tout de même stockées en parallèle dans un fichier `csv` "historique", mais qui n'a pas pour objectif d'être lu pour le trading.

Note : Les valeurs du bitcoin sont à présent récupérées en direct via une API (`CoinGecko`), puis un filtre d'aléatoire (informatique) y est appliqué pour obtenir le SRD-BTC.

Les données plus statiques, comme les identifiants et token clients, leurs différents soldes, suivent aussi le même stockage. À chaque nouvelle connexion, le serveur génère un fichier `csv` personnel où les informations sont inscrites, mais dès lors qu'une session est active, il crée également un tableau (en mémoire), où il charge tout ce dont le client a besoin (`Wallet`). Ainsi, l'ensemble de l'activité de ce dernier pourra être effectuée avec des lectures rapides.

Note : Ces informations sont sauvegardées à la destruction du tableau (`Wallet`) dans le fichier attitré au client, et chargées depuis ce fichier à chaque nouvelle connexion de ce même client.

Avant de passer aux interactions, et de parler un peu plus des accès concurrents côté client que nous avons dû gérer, voici ceux qui étaient côté serveur :

- **Accès Concurrent à `activeSessions`** : La map `activeSessions` est modifiée par le thread `acceptThread` (lors de l'ajout d'une nouvelle session dans `HandleClient`) et lue/modifiée par les threads `ClientSession` (lors de l'appel à `unregisterSession`). Un `std::mutex` (`sessionsMutex`) est donc indispensable pour protéger toutes les opérations sur cette map.

- **Accès Concurrent à `users`** : La map `users` est accédée par plusieurs threads `HandleClient` simultanément lorsqu'ils traitent des demandes d'authentification via `processAuthRequest`. L'accès en lecture (vérification de l'utilisateur) et en écriture (enregistrement d'un nouvel utilisateur) nécessite une protection par un `std::mutex` (`usersMutex`). La persistance (chargement/sauvegarde) doit également verrouiller ce mutex pendant l'opération fichier.

- **Gestion des Threads** : Le serveur doit gérer le cycle de vie de multiples threads (acceptation, et un thread par client). L'arrêt propre du serveur (`StopServer`) implique de signaler l'arrêt à ces threads (via des flags atomiques comme `acceptingConnections`) et d'attendre leur terminaison (`join`) pour éviter les crashs.

- **Partage d'Objets entre Threads** : Utiliser `std::shared_ptr` pour gérer la durée de vie d'objets comme `ClientSession` et `ServerConnection` qui sont partagés entre le thread serveur principal, le thread d'acceptation et les threads clients. L'héritage de `std::enable_shared_from_this` permet aux objets gérés par `shared_ptr` d'obtenir un `shared_ptr` valide vers eux-mêmes si nécessaire.

Passons maintenant aux mesures de performances propres au serveur :

2.4

sectionMesures de Performances

Les mesures de performance sont divisées en trois parties principales : le nombre de transactions par seconde, le nombre de connexions par seconde, et le nombre de clients maximum que le serveur peut gérer. Ces mesures nous permetterons d'estimer les performances du serveur implémenté.

- TPS : Transactions par seconde Nous réalisons un benchmark qui permet de déterminer le nombre de transactions que le serveur peut gérer par seconde. Pour cela, nous implémentons un seul client qui va réaliser plusieurs transactions d'affilé, pour différents nombre de transactions afin d'avoir plusieurs mesures. Les mesures montrent, pour des mesures entre 100 et 10 000 de transactions, un TPS entre 90 et 100. Ce qui nous donne un temps optimal moyen, du fait qu'il ne soit réalisé par un seul client, pour une transaction autour de 11 ms. Au dela de 4000 transactions, on remarque une baisse de transactions complétées bien que celles-ci soit prises en compte. Elles sont stockées dans la file d'attente et traduisent une véritable surcharge serveur.
- CPS : Connexions par seconde Nous réalisons ensuite un benchmark permettant de déterminer le nombre de connexions que le serveur peut gérer par seconde, et également le temps de connexion moyen pour une connexion. Afin de ne pas rajouter le facteur limitant du nombre de connexion maximal, ce benchmark déconnecte le client et libère la connexion SSL et CTX juste après toutes les étapes de connexion du client. Les mesures montrent un CPS entre 180 et 200 pour des mesures entre 1 000 et 20 000 connexions. Ce qui nous permet de déterminer un temps moyen de connexion de 5 ms. Cette mesure permet de déterminer que les connexions clients pourront être un facteur limitant dans le cas où un grand nombre de clients viendraient à se connecter avec un délai de moins de 5 ms entre chaque connexion. En effet, en dessous de 5 ms de délai entre les connexions clients on peut observer des connexions échouées, soit par mauvaise connexion TCP, soit par mauvaise connexion SSL, CTX. C'est donc un facteur qu'il faudra prendre en compte dans le serveur pour limiter les connexions afin d'éviter les surchages.
- Nombre de clients maximum connectés Nous réalisons enfin un benchmark pour mesurer le nombre de clients que le serveur peut supporter en simultané. Les mesures nous montrent qu'il nous est impossible d'aller au dela de 3000 clients, ce qui dépendra des machines sur lequel le serveur est à l'écoute. Nous avons également observé des baisse de capacité d'accueil client pouvant aller jusqu'à 1500 clients en simultané. Il faut noter que ce benchmark ne surcharge pas le nombre de transactions côté serveur, les capacités d'accueil pourraient en être impactées. Il se contente de connecter tous les clients les uns après les autres jusqu'à l'authentification (en passant par la connection SSL, CTX.

3 Les Interactions Clients

Voici sans doute la partie la plus ardue. Pour que notre architecture fonctionne, il a fallu concevoir et gérer plusieurs classes qui étaient là pour standardiser chaque requête, et ainsi permettre de faciliter les accès concurrents. Cette architecture est faite de deux piliers importants : les structures, les objets et fonctions *helpers* servant à récupérer telle ou telle donnée des données dans tel ou tel tableau, puis les fonctions **orchestres**. Celles-là étaient chargées de parser chaque message reçu, de définir une marche à suivre, d'appeler les bonnes fonctions, puis, chose essentielle, de **verrouiller** les objets critiques, auxquels plusieurs threads souhaitaient accéder. Nous parlerons de ces dernières en second, puisqu'elles sont finalement celles où il y a le moins à dire.

3.1 Les Composants Cœur du Trading et leur Gestion Multi-Threadée

Au cœur du système de trading se trouvent les mécanismes assurant le traitement des requêtes et la gestion des états financiers dans un environnement multi-threadé.

TransactionRequest : La Demande (L'Intention)

La **Demande de Transaction** (`TransactionRequest`) est une structure encapsulant l'intention initiale d'un client ou bot d'effectuer une opération de trading (`BUY`/`SELL`). Elle contient les paramètres de la demande : `clientId`, `RequestType`, `cryptoName`, `quantity`. Simple structure de données, elle est passée entre threads sans nécessiter de gestion de concurrence interne.

TransactionQueue : Le Moteur d'Exécution (Le Processeur Asynchrone)
La **File de Transactions** (`TransactionQueue`) est le processeur central des requêtes. Les `ClientSession`s soumettent leurs `TransactionRequest`s à la TQ pour un traitement asynchrone dans un thread dédié (`std::thread worker`). Ce modèle producteur-consommateur permet de ne pas bloquer les sessions clientes pendant l'exécution des transactions.

Wallet : L'état
Le **Portefeuille** (`Wallet`) gère l'état financier (`balances`, `transactionHistory`) d'un client. Ses données sont critiques et mutables dans un contexte multi-threadé. Chaque instance de `Wallet` possède son propre mutex. Ce mutex **doit** protéger toutes les opérations (lecture/écriture) sur les soldes et l'historique (`getBalance()`, `updateBalance()`, `addTransaction()`, `getTransactionHistory()`), ainsi que la persistance fichier (`saveToFile()`, `loadFromFile()`). Une méthode `std::mutex& getMutex()` permet d'obtenir le mutex spécifique à une instance pour effectuer des séquences d'opérations atomiques. Nous l'utilisons dans une des fonctions **orchestre**.

Transaction : Le Résultat (et l'enregistrement)
La transaction finale (**Transaction**) est un objet enregistrant les détails (`clientId`, type, quantité, prix, montant, frais, timestamp, statut `COMPLETED`/`FAILED`) et le résultat d'une opération traitée par la TQ. Une fois finalisée, elle est généralement immuable. Des membres statiques pour générer des IDs uniques ou pour le log global nécessitent aussi des mutex pour la thread safety.

Le Flux Global de Traitement des Requêtes : Le parcours d'une demande de transaction à travers la **TransactionQueue**, insistant sur la gestion des verrous.

1. Soumission de la requête par un thread **ClientSession** : Il verrouille, ajoute la requête à **Queue**, le signale, puis libère le mutex.
2. Le thread **txQueue.worker**, réveillé, détient le verrou, retire la requête de la file, puis le libère immédiatement (pour permettre d'autres soumissions).
3. Pour retrouver la session cliente correspondante, le thread de traitement **verrouille les informations des sessions actives**, cherche la bonne session par son identifiant, obtient une référence, puis **libère les informations des sessions actives**.
4. Le thread de traitement obtient ensuite l'accès au portefeuille du client associé à cette session.
5. Le thread de traitement **verrouille le portefeuille du client**. Ce verrou exclusif est maintenu pendant toute la durée des opérations nécessaires sur ce portefeuille pour garantir qu'elles soient faites sans interruption par d'autres actions.
6. Sous ce verrou du portefeuille sont effectuées les opérations nécessaires : vérification des soldes, calculs liés à la transaction, détermination du statut final (Réussie ou Échouée), mise à jour des soldes, ajout de la transaction à l'historique, et sauvegarde du portefeuille sur le disque.
7. Juste après avoir terminé les opérations sur le portefeuille sous verrou, le thread de traitement crée l'enregistrement final de la transaction, capturant tous les détails et le statut (générant un identifiant unique si nécessaire, avec un bref verrou pour cela).
8. Le thread de traitement **libère le verrou du portefeuille**. Le portefeuille est à nouveau accessible pour d'autres opérations.
9. L'enregistrement final de la transaction est ajouté à un historique global. Cela nécessite de **verrouiller l'accès à cet historique partagé** le temps de l'écriture pour éviter les conflits.
10. Le thread de traitement informe la session cliente concernée du résultat en lui transmettant l'enregistrement final de la transaction.
11. Dans le thread de la session cliente, les informations de l'enregistrement final sont utilisées pour formater et envoyer un message de notification au client.

Dans toute cette articulation, il est bon de rappeler qu'il y a également les **Composants Transversaux**. Ils incluent un gestionnaire de données globales (`Global`), un système de logging (`Logger`), voire même la `TransactionQueue` (en tant que service global).

Pour conclure et résumer sur l'importance des Mutex en Multi-Threading

- **Protection des Données Partagées** : Le rôle principal des mutex (`mtx`, `sessionMapMtx`, `walletMutex`, `counterMutex`, `persistenceMutex`) est de garantir que les sections de code qui accèdent ou modifient des données partagées (la file, la map des sessions, les soldes du Wallet, l'historique du Wallet, le compteur statique, le fichier de log) ne sont exécutées que par **un seul thread à la fois**. C'est la "**section critique**". `std::lock_guard` ou `std::unique_lock` sont les outils C++ standards pour gérer automatiquement l'acquisition et la libération du verrou.

- **Atomicité des Opérations** : `walletMutex` et son utilisation avec `lock_guard` dans `processRequest` sont essentiels pour rendre la séquence "vérifier solde → mettre à jour solde → ajouter historique → sauvegarder" **atomique** pour un portefeuille spécifique. Même si chaque méthode Wallet interne est thread-safe, la *séquence* d'appels ne l'est pas sans le verrouillage externe via `getMutex()`.
- **Prévention des Conditions de Course** : Sans mutex, on aurait des lectures "sales" (un thread lit un solde juste pendant qu'un autre est en train de le modifier), des écritures perdues (deux threads essaient d'écrire au même endroit, une écriture écrase l'autre), ou des structures de données corrompues (deux threads manipulent la file en même temps).

4 Le Client et la Logique du Bot de Trading

Maintenant que nous avons abordé ce qu'il se passait d'une façon plus *technique*, revenons sur les objectifs du projet. L'application permet l'interaction de l'utilisateur avec un marché simulé. En plus de cela Le composant `Bot` ajoute une couche d'automatisation basée sur une stratégie de trading.

4.1 Rôle du Client

Comme vu brièvement au départ, le client est l'interface ou le point de commencement de l'interaction avec le serveur. Ses fonctions incluent :

- **Connexion au Serveur** : Établir une connexion réseau avec le serveur.
- **Authentification/Identification** : S'identifier auprès du serveur (via `clientId`).
- **Envoi de Requêtes** : Formater et envoyer des requêtes au serveur (demandes d'information ou d'opération).
- **Réception et Affichage des Réponses** : Recevoir les réponses du serveur et les présenter à l'utilisateur ou les traiter.

Une implémentation que nous avons faite en plus lors de ce semestre, est de lui permettre d'acheter ou de vendre lui-même. Bien sûr, il lui reste possible de faire appel à une stratégie automatisée que nous appelons "bot". Parlons d'abord un peu de cette stratégie avant d'observer ses performances.

4.2 Le Bot de Trading (`Bot`)

Le `Bot` incarne la logique de trading automatique associée à un `clientId` et opérant sur le `Wallet` correspondant.

- **Association au Client/Portefeuille** : Chaque instance de `Bot` est liée à un client spécifique et a accès à son portefeuille (`std::shared_ptr<Wallet>`).
- **Stratégie de Trading** : Le bot implémente une stratégie de trading (identifiée comme basée sur les Bandes de Bollinger, avec des paramètres comme `bollingerPeriod` et `bollingerK`).
- **Traitement des Données de Marché** : Le bot reçoit les mises à jour des prix du marché (via la `BotSession` qui les obtient du serveur/Global) et les utilise dans sa méthode `processLatestPrice` pour analyser la situation. **Note** : Dans notre architecture finalisée, le bot accède aux données via des mécanismes globaux ou partagés, pas nécessairement via une `BotSession` distincte. Le flux des données de prix vers les bots actifs est un point clé d'implémentation multi-threadée.
- **Prise de Décision** : Basé sur l'analyse (calculs SMA, Bandes de Bollinger) et son état actuel (`PositionState` : `NONE`, `LONG`, `SHORT`), le bot décide d'une action de trading (`TradingAction` : `BUY`, `SELL`, `CLOSE_LONG`, `CLOSE_SHORT`, `HOLD`).
- **Soumission des Transactions** : Lorsqu'une action de trading est décidée (différente de `HOLD`), le bot initie une demande de transaction (`TransactionRequest`), qui est ensuite transmise à un service d'exécution centralisé (la `TransactionQueue` du serveur) pour traitement.
- **Mise à Jour de l'État** : Le bot est notifié par le système centralisé de l'achèvement ou de l'échec d'une transaction (via un appel de la `TransactionQueue` vers la `ClientSession`) afin de mettre à jour son état de position (`currentState`) et son prix d'entrée (`entryPrice`).

La logique pure du bot est donc encapsulée dans des méthodes comme `calculateSMA`, `calculateStdDev`, `calculateBands`, et le cœur de la décision dans `processLatestPrice`.

4.3 Interaction entre Client, Bot et Serveur

L'interaction typique se déroule comme suit :

1. Le client se connecte au serveur.
2. Une `ClientSession` est créée sur le serveur pour ce client, gérant une instance de `Wallet` et potentiellement une instance de `Bot`. (La "BotSession" mentionnée précédemment est probablement intégrée dans la ClientSession ou est un concept distinct pour la gestion du bot).
3. Le serveur (ou un composant dédié géré par le serveur) obtient les prix du marché (par exemple, via une API comme CoinGecko) et les rend disponibles (`Global`).
4. Les mises à jour de prix sont transmises aux `ClientSession`s actives qui gèrent un bot (via un mécanisme d'abonnement ou de notification depuis le flux de prix global).
5. Chaque `ClientSession` passe le prix à son `Bot` via `processLatestPrice`.
6. Le `Bot` analyse le prix et sa stratégie décide d'une `TradingAction`.
7. Si l'action est une opération (BUY/SELL/CLOSE), le `Bot` initie une `TransactionRequest` et la soumet à la `TransactionQueue` du serveur.
8. Un thread d'exécution de transaction (le `worker` de la `TransactionQueue`) traite la demande : il **verrouille** le `Wallet` associé, vérifie le solde, met à jour les soldes (`updateBalance()`, `addTransaction()`), et crée un objet `Transaction` final avec le statut `COMPLETED` ou `FAILED`.
9. La `ClientSession` associée est notifiée par la `TransactionQueue` de l'achèvement de la transaction (via un appel spécifique, potentiellement `applyTransactionRequest`).
10. Le `Bot` (géré par la `ClientSession`) met à jour son état interne en fonction du résultat de la transaction notifié par la session.

Ce cycle permet au bot de réagir aux conditions du marché et d'exécuter des transactions via le système centralisé du serveur.

4.4 Performances du Bot

Ces performances sont mesurées sur plusieurs *runs* d'environ 1000 données.

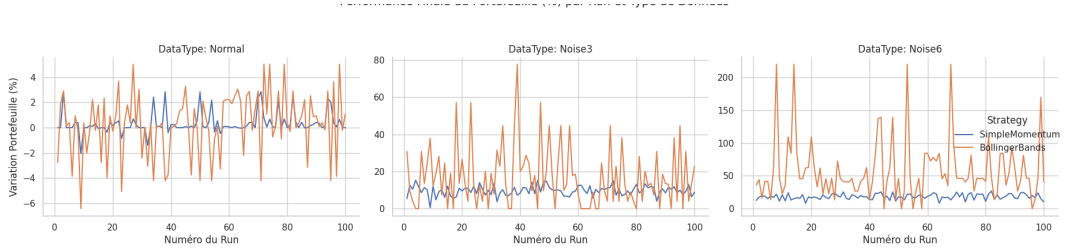


Figure 1: Comparaison d'une méthode de Bollinger avec une méthode d'investissement plus régulière

Nous pouvons aisément observer le gain significatif lorsque l'on utilise la méthode de Bollinger, surtout à mesure que la composante d'aléatoire grandit. Nous concluons donc quant à son efficacité dans le cadre de notre monnaie SRD-BTC.

5 Quelques graphiques sur les performance

5.1 Profiling

Nous commencerons cette dernière partie par une analyse des métriques que nous avons jugées les plus intéressantes. En effet, lorsque nous avons débuté notre travail d'optimisation, beaucoup d'options s'offraient à nous. Il nous fallait d'abord mener une analyse plus poussée de notre code afin de repérer les endroits critiques, les parties indispensables à retravailler. De façon assez naturelle, lorsque l'on passe sur un modèle multi-threadé, nous avons commencé par nous intéresser à l'accès aux sections critiques. Ce sont autour d'elles que vont s'articuler les principales failles potentielles de notre code. Que se passe-t-il lors d'une concurrence entre threads pour l'accès à une valeur par exemple ? Ou encore si une quantité importante de clients souhaitent se connecter dans un temps réduit ? Autant de situations qui nous ont paru critiques, et sur lesquelles nous avons travaillé.

Gestion des requêtes : Solution, la file de transactions (`TransactionQueue`). Une requête est faite, elle est directement envoyée dans une file d'attente avec un timestamp et peut-être un pricestamp pour éviter d'aller récupérer le prix dans un tableau.

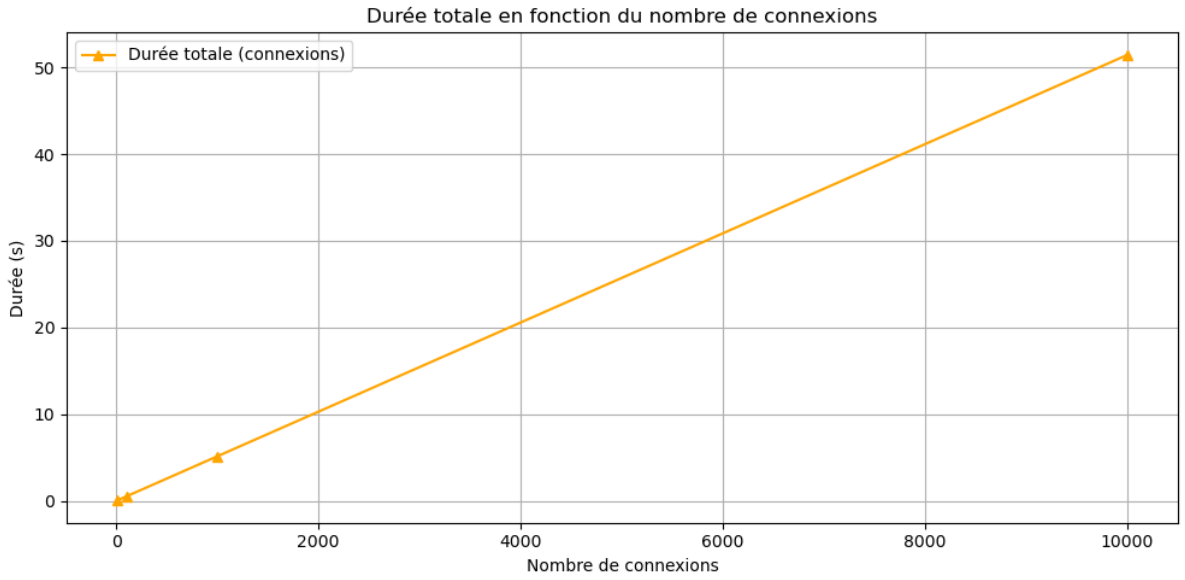
Gestion des connexions : [Ce point semble incomplet dans le texte original]

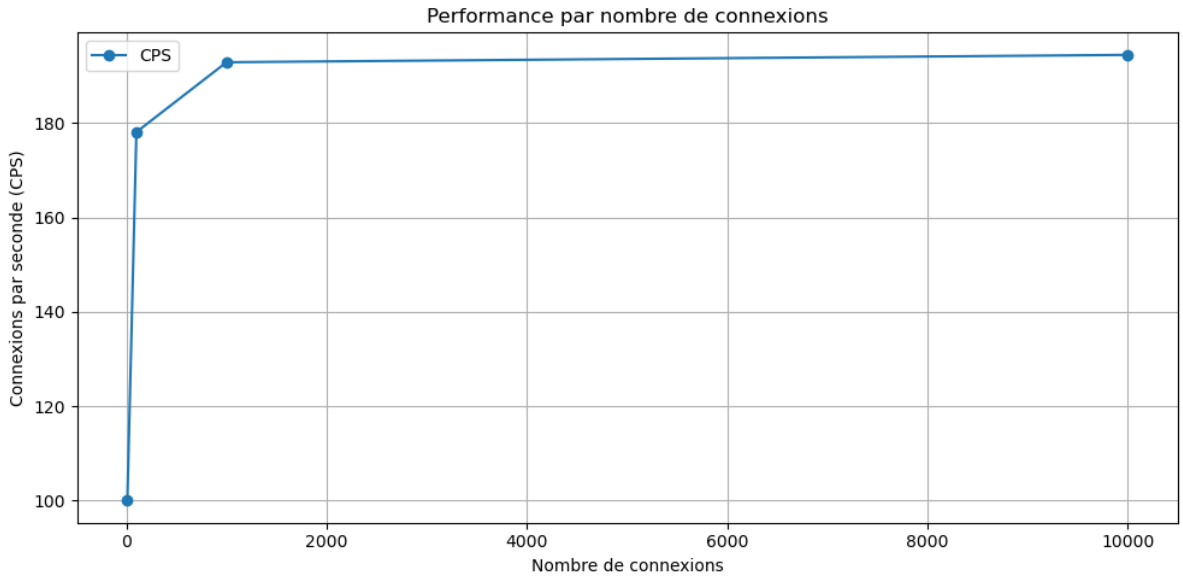
Accès mémoire : Solution, passage de fichiers `csv` à des tableaux plus petits (une journée de valeurs par exemple) pour un échange plus rapide de données en mémoire vive.

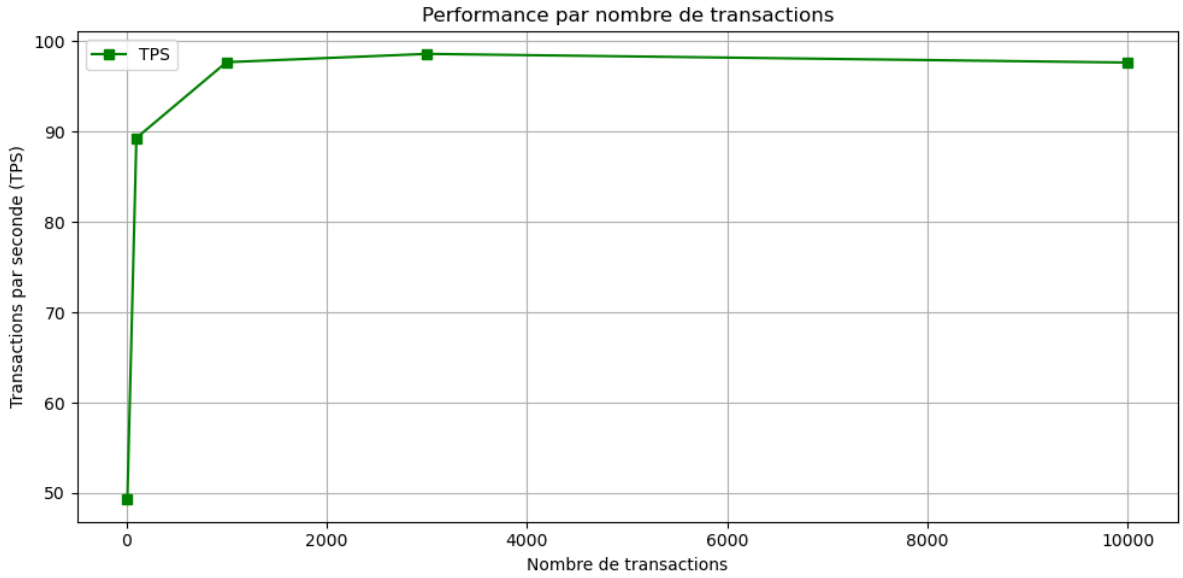
Performance des méthodes de trading : Passer d'un algorithme simple à un algorithme plus poussé. Mesurer les performances.

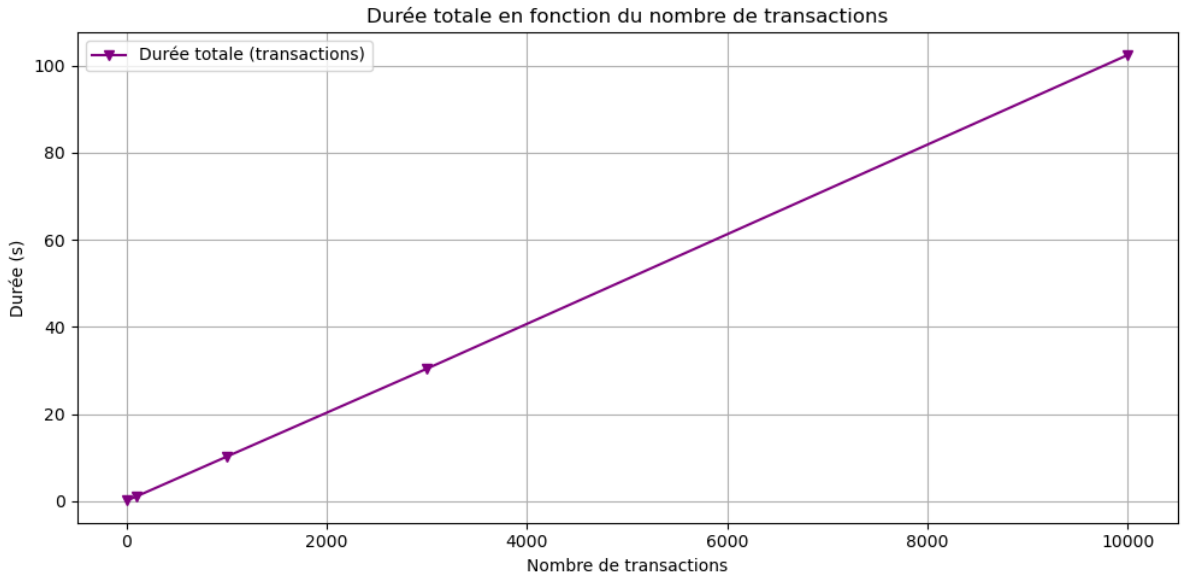
5.2 Résultats des Benchmarks

Dans cette section, nous présentons les résultats des benchmarks réalisés pour évaluer les performances du serveur, y compris les graphiques illustrant le TPS et le CPS.




Performance par nombre de connexions
CPS
Connexions par seconde (CPS)
100
120
140
160
180
0
2000
4000
6000
8000
10000
Nombre de connexions


Performance par nombre de transactions
TPS
Transactions par seconde (TPS)
50
60
70
80
90
100
0
2000
4000
6000
8000
10000
Nombre de transactions



6 Conclusion

Le projet PPN_CTS présente une architecture modulaire bien définie, avec une séparation des responsabilités entre le serveur (coordination, communication), les sessions clients (gestion par client), les bots (logique de trading), les portefeuilles (gestion financière) et les transactions (modélisation des opérations). L'utilisation de mécanismes de synchronisation (`std::mutex`, `std::lock_guard`) et de structures de données adaptées (`std::map`, `std::vector`, `std::queue`) vise à assurer un fonctionnement thread-safe.